

Lecture 7

Trees, Binary trees data structures

Trees

- A tree is composed of a collection of *nodes*, where each node has some associated data and a set of *children*.
- Node's children are those nodes that appear immediately beneath the node itself
- A node's *parent* is the node immediately above it
- A tree's *root* is the single node that contains no parent
- Examples for data where trees is the data structure of choice:
Organization charts, file systems
folders and files or anywhere
hierarchy is needed

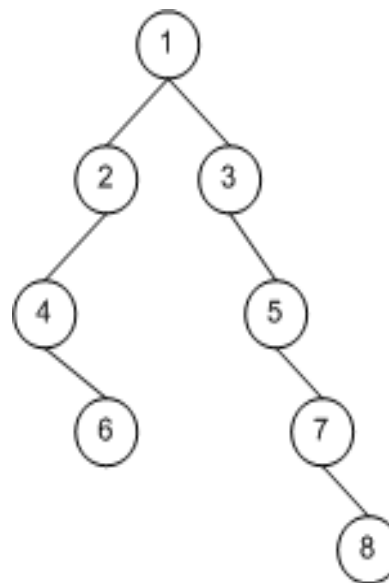


Tree characteristics

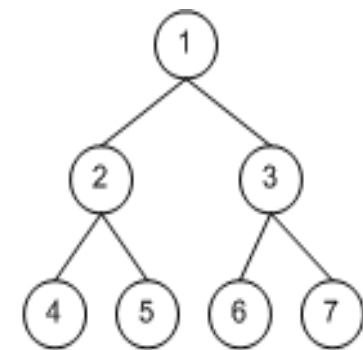
- All trees exhibit the following properties:
 - There is precisely one root.
 - All nodes except the root have precisely one parent.
 - There are no *cycles*. That is, starting at any given node, there is not some path that can take you back to the starting node. The first two properties—that there exists one root and that all nodes except the root have one parent—guarantee that no cycles exist.

Binary trees

- A binary tree is a special kind of tree
- One in which all nodes have at most two children.
- For a given node in a binary tree, the first child is referred to as the *left* child, while the second child is referred to as the *right* child.
- The shown two trees are binary trees
- Nodes that have no children are referred to as *leaf nodes*. Nodes that have one or two children are referred to as *internal nodes*.



(a)



(b)

Tree implementation, Node and BinaryTreeNode

```
public class Node<T>
{
    // Private member-variables
    private T data;
    private List<Node<T>> neighbors = null;
    public Node() { }
    public Node(T data) : this(data, null) { }
    public Node(T data, List<Node<T>> neighbors)
    {
        this.data = data;
        this.neighbors = neighbors;
    }
    public T Value
    {
        get{ return data; }
        set{ data = value; }
    }
    protected List<Node<T>> Neighbors
    {
        get{ return neighbors; }
        set{ neighbors = value; }
    }
}
```

```
public class BinaryTreeNode<T> : Node<T>
{
    // the following constructor is avoided because it neither creates nor initializes the Neighbors list
    //public BinaryTreeNode() : base() { }
    public BinaryTreeNode() : this(default(T), null, null) { }
    public BinaryTreeNode(T data) : this(data, null, null) { }
    public BinaryTreeNode(T data, BinaryTreeNode<T> left, BinaryTreeNode<T> right)
    {
        base.Value = data;
        this.Neighbors = new List<Node<T>>(2);
        this.Neighbors.Add(left); this.Neighbors.Add(right);
    }
    public BinaryTreeNode<T> Left
    {
        get{ return (BinaryTreeNode<T>)base.Neighbors[0]; }
        set{ this.Neighbors[0] = value; }
    }
    public BinaryTreeNode<T> Right
    {
        get{ return (BinaryTreeNode<T>)base.Neighbors[1]; }
        set{ this.Neighbors[1] = value; }
    }
}
```

Node is a general node to represent a graph node or a tree node
BinaryTreeNode is a node for a binary tree

Tree implementation, BinaryTreeNode

```
public class BinaryTreeNode<T>
{
    private BinaryTreeNode<T> left, right;
    private T data;
    public BinaryTreeNode() : this(default(T), null, null) { }
    public BinaryTreeNode(T data) : this(data, null, null) { }
    public BinaryTreeNode(T data, BinaryTreeNode<T> left, BinaryTreeNode<T> right)
    {
        this.Value = data;
        this.left = left;
        this.right = right;
    }
    public T Value
    {
        get { return data; }
        set { this.data = value; }
    }
    public BinaryTreeNode<T> Left
    {
        get { return left; }
        set { this.left = value; }
    }
    public BinaryTreeNode<T> Right
    {
        get { return right; }
        set { this.right = value; }
    }
}
```

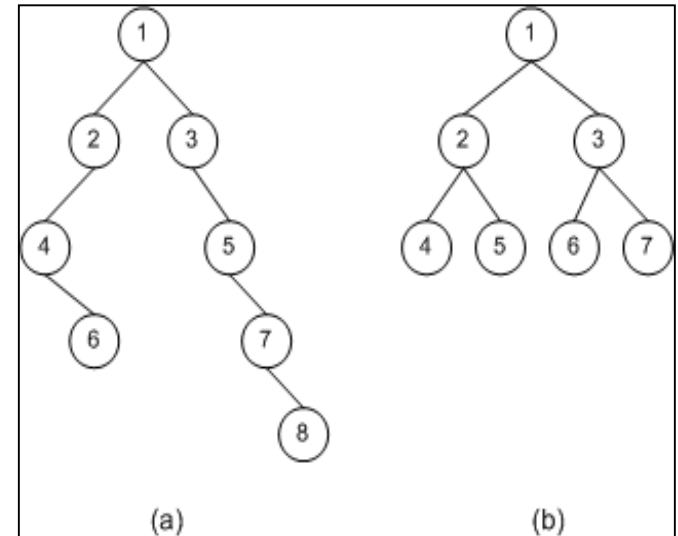
A simple Binary tree node, if we are interested only in binary tree

Binary tree class

```
public class BinaryTree<T>
{
    private BinaryTreeNode<T> root;
    public BinaryTree()
    {
        root = null;
    }
    public virtual void Clear()
    {
        root = null;
    }
    public BinaryTreeNode<T> Root
    {
        get
        {
            return root;
        }
        set
        {
            root = value;
        }
    }
}
```

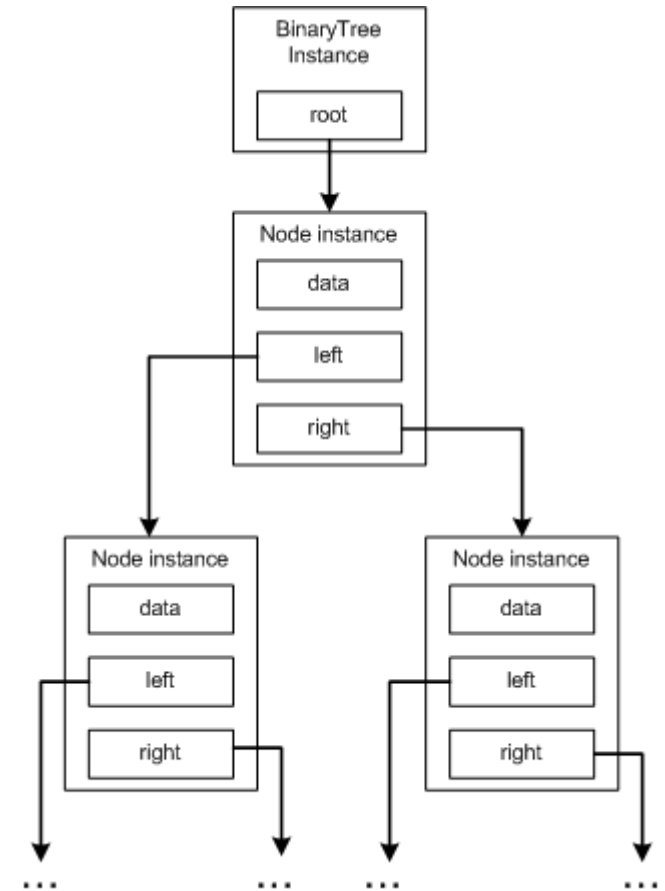
Using the BinaryTree class

```
class Example1
{
    static void Main(string[] args)
    {
        BinaryTree<int> btreeA = new BinaryTree<int>();
        btreeA.Root = new BinaryTreeNode<int>(1);
        btreeA.Root.Left = new BinaryTreeNode<int>(2);
        btreeA.Root.Right = new BinaryTreeNode<int>(3);
        btreeA.Root.Left.Left = new BinaryTreeNode<int>(4);
        btreeA.Root.Right.Right = new BinaryTreeNode<int>(5);
        btreeA.Root.Left.Left.Right = new BinaryTreeNode<int>(6);
        btreeA.Root.Right.Right.Right = new BinaryTreeNode<int>(7);
        btreeA.Root.Right.Right.Right.Right = new BinaryTreeNode<int>(8);
        BinaryTree<int> btreeB = new BinaryTree<int>();
        btreeB.Root = new BinaryTreeNode<int>(1);
        btreeB.Root.Left = new BinaryTreeNode<int>(2);
        btreeB.Root.Right = new BinaryTreeNode<int>(3);
        btreeB.Root.Left.Left = new BinaryTreeNode<int>(4);
        btreeB.Root.Left.Right = new BinaryTreeNode<int>(5);
        btreeB.Root.Right.Left = new BinaryTreeNode<int>(6);
        btreeB.Root.Right.Right = new BinaryTreeNode<int>(7);
        Console.WriteLine("Press any key to continue:");
        Console.Read();
    }
}
```



Tree access time

- Array's elements are stored in a contiguous block of memory. By doing so, arrays exhibit constant-time lookups (Direct access). This is also true for queues and stacks if implemented using arrays or ArrayLists (or the generic equivalent Lists)
- Binary trees, however, are not stored contiguously in memory. The BinaryTree class instance has a reference to the root BinaryTreeNode class instance. The root BinaryTreeNode class instance has references to its left and right child BinaryTreeNode instances; these child instances have references to their child instances, and so on.
- The point is, the various BinaryTreeNode instances that makeup a binary tree can be scattered throughout the CLR managed heap. They are not necessarily contiguous, as are the elements of an array.
- So, Direct access is not possible with trees, and for reaching an element, we must search all the tree for that element which require a linear time in the worst case



Report Discussion

Last lecture report:

- DOTNET Queue class methods and properties
- The queue role in Operating systems

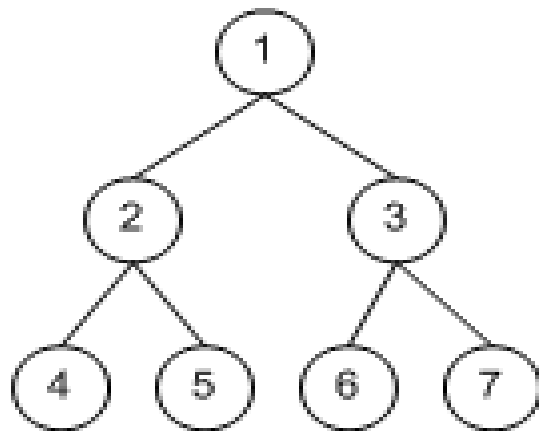
New report:

- Does the order in which the nodes added to the binary tree affect the search time for a node with a given value?
- Give some demonstrating examples

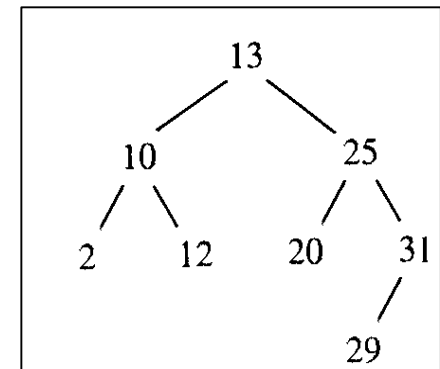
Traversing a Tree

- Tree traversal specifies only one condition—visiting each node only one time—but it does not specify the order in which the nodes are visited
- There are $n!$ way to traverse a tree with n nodes
- Only some regular ways are common and implemented
 - Breadth-First Traversal
 - Depth-First Traversal
 - Three common Depth-First traversal orders for binary tree
 - PreOrder Traversal
 - InOrder Traversal
 - PostOrder Traversal

Binary Tree Traversal



PreOrder Traversal	1, 2, 4, 5, 3, 6, 7
InOrder Traversal	4, 2, 5, 1, 6, 3, 7
PostOrder Traversal	4, 5, 2, 6, 7, 3, 1



Breadth-First:
13, 10, 25, 2, 12, 20, 31, 29

Traversing binary Tree implementation

Recursive functions are often ideal for visualizing an algorithm, as they can often elegantly describe an algorithm in a few short lines of code. However, recursive functions are usually sub-optimal when compared to iterating through a data structure's elements from the performance and storage point of view

```
static void PreorderTraversal(BinaryTreeNode<int> current)
{
    if (current != null)
    {
        // Output the value of the current node
        Console.WriteLine(current.Value);
        // Recursively print the left and right children
        PreorderTraversal(current.Left);
        PreorderTraversal(current.Right);
    }
}

static void InorderTraversal(BinaryTreeNode<int> current)
{
    if (current != null)
    {
        // Visit the left child...
        InorderTraversal(current.Left);
        // Output the value of the current node
        Console.WriteLine(current.Value);
        // Visit the right child...
        InorderTraversal(current.Right);
    }
}

static void PostorderTraversal(BinaryTreeNode<int> current)
{
    if (current != null)
    {
        // Visit the left child...
        PostorderTraversal(current.Left);
        // Visit the right child...
        PostorderTraversal(current.Right);
        // Output the value of the current node
        Console.WriteLine(current.Value);
    }
}
```